

J/A 998218

日 本 国 特 許 庁
PATENT OFFICE
JAPANESE GOVERNMENT

JCS25 U.S. PTO
09/490582
01/25/00

別紙添付の書類に記載されている事項は下記の出願書類に記載されて
る事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed
with this Office.

出 願 年 月 日
Date of Application:

1999年 1月27日

願 番 号
Application Number:

平成11年特許願第017943号

願 人
Applicant(s):

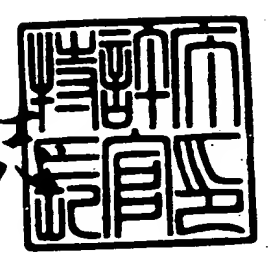
インターナショナル・ビジネス・マシーンズ・コーポレイシ
ョン

BEST AVAILABLE COPY

1999年 3月19日

特許庁長官
Commissioner,
Patent Office

伴佐山 建志



出証番号 出証特平11-3016490

【書類名】 特許願
【整理番号】 JA998218
【提出日】 平成11年 1月27日
【あて先】 特許庁長官 伊佐山 建志 殿
【国際特許分類】 G06F 9/44
【発明の名称】 多次元配列オブジェクトの処理方法及び装置
【請求項の数】 9
【発明者】

【住所又は居所】 神奈川県大和市下鶴間1-6-23番地1-4 日本アイ・ピー・エム株式会社 東京基礎研究所内

【氏名】 古関 聰

【発明者】

【住所又は居所】 神奈川県大和市下鶴間1-6-23番地1-4 日本アイ・ピー・エム株式会社 東京基礎研究所内

【氏名】 稲垣 達氏

【発明者】

【住所又は居所】 神奈川県大和市下鶴間1-6-23番地1-4 日本アイ・ピー・エム株式会社 東京基礎研究所内

【氏名】 小松 秀昭

【特許出願人】

【識別番号】 390009531

【氏名又は名称】 インターナショナル・ビジネス・マシーンズ・コーポレーション

【氏名又は名称原語表記】 INTERNATIONAL BUSINESS MACHINES CORPORATION
N

【代理人】

【識別番号】 100086243

【弁理士】

【氏名又は名称】 坂口 博

【代理人】

【識別番号】 100091568

【弁理士】

【氏名又は名称】 市位 嘉宏

【復代理人】

【識別番号】 100059258

【弁理士】

【氏名又は名称】 杉村 暁秀

【復代理人】

【識別番号】 100072051

【弁理士】

【氏名又は名称】 杉村 興作

【復代理人】

【識別番号】 100098383

【弁理士】

【氏名又は名称】 杉村 純子

【手数料の表示】

【予納台帳番号】 015093

【納付金額】 21,000円

【提出物件の目録】

【物件名】 明細書 1

【物件名】 図面 1

【物件名】 要約書 1

【包括委任状番号】 9304391

【包括委任状番号】 9304392

【ブルーフの要否】 要

【書類名】 明細書

【発明の名称】 多次元配列オブジェクトの処理方法及び装置

【特許請求の範囲】

【請求項 1】 多次元配列を構成している配列オブジェクトからなる多次元配列オブジェクトに対し、その多次元配列の要素に対する処理を最適化してもよいことを示す処理最適化可能フラグを管理し、前記処理最適化可能フラグの状態に対応する機械コードを実行することを特徴とする多次元配列オブジェクトの処理方法。

【請求項 2】 前記処理最適化可能フラグは、定められた条件が満たされなくなった場合に反転されることを特徴とする請求項 1 記載の多次元配列オブジェクトの処理方法。

【請求項 3】 前記定められた条件が、多次元配列オブジェクトの基底配列が連続したメモリ領域に割り当てられていることである請求項 2 記載の多次元配列オブジェクトの処理方法。

【請求項 4】 前記多次元配列オブジェクトを処理する部分の機械コードは、前記処理最適化可能フラグの状態に対応して、前記定められた条件に従って最適化を行った機械コード又は最適化を行っていない機械コードとなる請求項 2 記載の多次元配列オブジェクトの処理方法。

【請求項 5】 前記多次元配列オブジェクトへの書き込み時に、前記定められた条件が満たされているか否か判断する請求項 2 記載の多次元配列オブジェクトの処理方法。

【請求項 6】 前記多次元配列オブジェクトの生成時に、前記定められた条件が満たされている場合には、生成された多次元配列オブジェクトに対し処理最適化可能フラグを立てる請求項 2 記載の多次元配列オブジェクトの処理方法。

【請求項 7】 前記多次元配列オブジェクトのマルチスレッド処理の可能性がある場合には、最適化コードを実行中、前記多次元配列に対するダミーの参照をスタック上に格納する機械コードを生成する請求項 1 記載の多次元配列オブジェクトの処理方法。

【請求項 8】 多次元配列を構成している配列オブジェクトからなる多次元配列

オブジェクトに対し、その多次元配列の要素に対する処理を最適化してもよいことを示す処理最適化可能フラグを管理する手段と、前記処理最適化可能フラグの状態に対応する機械コードを実行する手段とを有する多次元配列オブジェクトの処理装置。

【請求項 9】 多次元配列を構成している配列オブジェクトからなる多次元配列オブジェクトに対し、その多次元配列の要素に対する処理を最適化してもよいことを示す処理最適化可能フラグを管理する手段と、前記処理最適化可能フラグの状態に対応する機械コードを実行する機能を実現するためのプログラムを格納した記憶媒体。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】

本発明は、多次元配列が配列オブジェクトの配列で実現された言語（例えば Java (Sun Microsystems Inc. の商標)）における多次元配列オブジェクトの処理方法及び装置に関するものである。

【0002】

【従来の技術】

多次元配列の要素に対する処理を並列化、ベクトル化することで、処理速度の飛躍的な向上を図ることができる。特に、IA64、PowerPC 等の実行ユニットを複数持ったプロセッサでは、このような最適化による処理速度の向上の効果は著しい。これらの最適化を行うためには、多次元配列要素の処理間の依存関係や、配列の要素のロケーションの解析が必要である。しかしながら、Javaのような多次元配列を配列オブジェクトの配列で実現している言語では、多次元配列の要素構成が実行時に変化してしまう可能性があり、そのような解析を行うのが困難である。

【0003】

この問題に対しては、従来大きく分けて二つの解決法があった。まず、ある処理によってアクセスされる多次元配列要素のアドレスを実行時に走査して解析を行う方法が知られている。また、言語仕様の拡張、あるいは独自のライブラリの

導入によって、要素構成が実行時に変化しない多次元配列を実現する方法も知られている。これらの方法を用いることで、たしかに多次元配列に対する処理の最適化は可能になる。しかしながら、これらの方法では、以下に述べるような様々な問題点が存在していた。

【0004】

【発明が解決しようとする課題】

まず、前者の方法は、アドレスの走査に時間がかかりすぎる上、アドレス走査が開始された時点から配列の処理が終了する時点までガーベッジコレクタを停止させなくてはならないという問題があった。特に、アドレス走査にかかる時間の問題は、Just-in-Timeコンパイラで処理させることが一般的になりつつあるJavaでは致命的であり、現実的な解決法ではない。

【0005】

一方、後者の方法に関しては、Sun Microsystems社によって、一次元配列としてメモリが割り当てられた領域を多次元配列としてアクセスするようなクラスを用いる提案がなされている。これは、まず以下のようなクラスを用意する。

```
final class FloatMatrix2D {  
    private final float m;  
    private final int cols;  
    public FloatMatrix2D(int c,int r)  
    {  
        cols=c;  
        m=new float c*r ;  
    }  
    public float get(int c,int r)  
    {  
        return m cols*r+c ;  
    }  
}
```

【0006】

ここで、メソッドインライニングとオペレータオーバーローディングを利用して、

```
FloatMatrix2D f=new FloatMatrix2D(cols,rows)
```

```
f i1,j1 =f i2,j2 +4;
```

のように記述された式を、

```
f i1,j1 =f i2,j2 +4;
```

↓

```
T=f.get(i2,j2)+4;
```

```
f.put(i1,j1,T);
```

↓

```
T=f cols*j2+i2 +4;
```

```
f cols*j1+i1 =T;
```

のように扱う方式である。このような方式を用いることで、オブジェクト配列の参照を経ずに直接要素にアクセスすることができる。また、オブジェクト配列の参照時に必要なインデックスのチェックもする必要がない。しかしながら、この方式では、並列化あるいは最適化に必要な解析の適用が困難になるという問題がある。

【0007】

すなわち、上述の方式においては、バイトコード化されたソースに何の情報もない以上、配列アクセス間の依存関係解析は一次元に変換されたインデックスにおける解析以上のことはできない。したがって、多次元性を考慮した並列性の検出は非常に困難になり、多くの場合において並列化の適用は期待できなくなる。例えば、

```
for j
```

```
  for i
```

```
    f i,j =f 2*i-1,j ;
```

のようなプログラムでは、j 方向の依存性がないことは簡単に解析できるので、最内ループを j 方向にスキューニングしたり、j 方向にループをアンローリングすることで、大きな並列性を得ることができる。ところが、

$$f \text{ cols} * j + i = f \text{ cols} * j + 2 * i - 1$$

のようにバイトコードが変換されてしまうと、

$$\text{cols} * j_1 + i_1 = \text{cols} * j_2 + 2 * i_2 - 1$$

に整数解があるかどうかを求め、その上で、j 方向に関して依存がないかどうかを判定しなければならない。これは、単純な解析問題を NP 完全問題にしてしまっていることを意味する。一般的に言っても、多次元の情報が一次元に縮退することで問題がより解決困難になると考えられ、Sun Microsystems 社の方式は並列化のための依存解析に適した方法とは言えない。

【0008】

また、他の方法として、言語の仕様を変更または拡張して依存解析を可能にする方法がある。例えば、Fortran のように、多次元配列を言語に導入する方法が考えられる。あるいは、上述の配列を単純に一次元化する方式においても、次のような拡張仕様を加えることで依存解析を可能とすることが考えられる。すなわち、先ほどの例で用いた

$$f \text{ cols} * j + i = f \text{ cols} * j + 2 * i - 1$$

のような式において、言語の仕様を拡張し、この配列が 2 次元であることと、f、cols がカラム数であることをコンパイラが利用できるようにすれば、インデックスの比較を多次元に展開して解析を容易にすることができる。例えば、この例では、問題を、

$(\text{cols} * j + i) \div \text{cols}$ と

$(\text{cols} * j + 2 * i - 1) \div \text{cols}$ の解析

$(\text{cols} * j + i) \bmod \text{cols}$ と

$(\text{cols} * j + 2 * i - 1) \bmod \text{cols}$ の解析

に置き換えることができ、j 方向に並列性があることを簡単に解析可能である。しかしながら、このような仕様変更や拡張のコストは非常に大きい。特に、Java に関しては、現在 Java が全世界的に広く普及している現状を鑑みると、新しい配列オブジェクトを追加するような仕様拡張や、特定のライブラリに依存した仕様拡張を行って効果を上げることは容易ではない。

【0009】

また、言語に仕様を拡張せずに、算術式の評価木 (Expression Tree) を解析する方法も考えられるが、この方式は次元数を特定するためのサポートが必要である上、共通部分式の削除等により Expression Tree が変形されると解析が困難になってしまうという問題があった。

【0010】

本発明の目的は上述した課題を解消して、仕様の変更を伴うことなく、多次元配列の処理速度の向上を可能とする多次元配列オブジェクトの処理方法及び装置を提供しようとするものである。

【0011】

【課題を解決するための手段】

本発明は、多次元配列が配列オブジェクトの配列で実現された言語 (例えば Java) における多次元配列オブジェクトの処理方法が対象となる。そして、多次元配列を構成している配列オブジェクトからなる多次元配列オブジェクトに、その多次元配列の要素に対する処理を最適化してもよいことを示す処理最適化可能フラグを付加情報として追加する。処理最適化可能フラグは、記憶装置 (例えばメインメモリ) に格納される。その後、処理最適化可能フラグの状態に対応した機械コードの実行を行う。

【0012】

本発明では、多次元配列の処理最適化可能フラグを導入し、多次元配列の要素構成の変更を監視しながら、同フラグが立っている配列に対して最適化された処理を実行してよいことを保証する。これにより、仕様の変更を伴うことなく、多次元配列の処理最適化可能性を実行時に高速に判断し、処理最適化可能性があると判断された多次元配列を最適化されたコードで処理することにより、プログラムの実行速度を向上させることができる。その結果、従来は困難であった配列オブジェクトの配列で実現された多次元配列の処理の高速化が可能になる。

【0013】

【発明の実施の形態】

本発明の対象となる Java のような言語では、多次元配列は配列オブジェクトの配列で実現されている。これらのオブジェクト配列 (配列オブジェクトの配列)

の内容は自由に書き換えることができるので、多次元配列の要素構成が実行時に変化してしまう可能性がある。したがって、多次元配列の各要素に対する処理を最適化してよいかどうかを判断するのは非常に難しい。一方、実際には、多くの多次元配列は最適化が可能であるような要素構成で生成され、また、オブジェクト配列の書き換えによる処理最適化可能性の破壊はめったに起こらない。したがって、多くの場合、多次元配列処理の最適化が可能であるにも関わらず、最適化を行えないという状態になっている。本発明では、以上の事実を前提として、多次元配列を構成している配列オブジェクトからなる多次元配列オブジェクトに、その多次元配列の要素に対する処理を最適化してもよいことを示す処理最適化可能フラグを付加情報として追加し、処理最適化可能フラグの状態に対応した機械コードの実行を行う。

【0014】

好適な実施態様として、処理最適化可能フラグの状態に対応した機械コードは、実行時に動的に生成しても、予め用意しておいてもよい。なお、処理最適化可能フラグは、定められた条件が満たされなくなった場合に反転されるように構成される。この定められた条件は、例えば、多次元配列オブジェクトの基底配列が連続したメモリ領域に割り当てられているということである。また、好適な実施態様として、多次元配列オブジェクトを処理する部分の機械コードは、処理最適化可能フラグの状態に対応して、定められた条件に従って最適化を行った機械コード又は最適化を行っていない機械コードのいずれかとなる。なお、多次元配列オブジェクトへの書き込み時に、定められた条件が満たされているか否か判断され、定められた条件が満たされていない場合には、処理最適化可能フラグが反転されるように構成される。一方、多次元配列オブジェクトの生成時に、定められた条件が満たされている場合には、生成された多次元配列オブジェクトに対し処理最適化可能フラグを立てておくよう構成する。

【0015】

Javaのような言語ではありえることであるが、多次元配列オブジェクトのマルチスレッド処理の可能性がある場合には、最適化コードを実行中、多次元配列に対するダミーの参照をスタック上に格納する機械コードを生成し、ガーベッ

ジコレクタによる回収を防止する。

【0016】

次に、本発明の多次元配列オブジェクトの処理方法をJava Just-in-Time コンパイラにおいて実施する例について説明する。

まず、Javaを用いた環境における、本発明の装置構成を図4を用いて説明する。サーバ・コンピュータ1及びクライアント・コンピュータ5はネットワーク3を介して接続されている。クライアント・コンピュータ5は、Java VM (Virtual Machine) 52及びOS (Operating System) 53及びハードウェア (CPU及びメモリを含む) 55を含む。さらに、Java VM 52は、Java インタプリタ54又はJava JITコンパイラ56を含む。インタプリタ54及びJITコンパイラ56の両者を有している場合もある。なお、クライアント・コンピュータ5は、通常のコンピュータの他、メモリの大きさが小さかったり、ハードディスク等の補助記憶装置を含まないような、いわゆるネットワーク・コンピュータや情報家電の場合もある。

【0017】

サーバ・コンピュータ1では、Javaソースコード10は、Javaコンパイラ12によりコンパイルされる。このコンパイルの結果が、バイトコード14である。このバイトコード14は、ネットワーク3を介してクライアント・コンピュータ5に送信される。バイトコード14は、クライアント・コンピュータ5内のWWWブラウザ (World Wide Web Browser) などに設けられたJava仮想マシン (Java VM) 52にとってネイティブ・コードであり、実際ハードウェア55のCPUにて実行する場合には、Javaインタプリタ54や、Java JITコンパイラ56を用いる。インタプリタ54は、実行時にバイトコード14をデコードし、命令ごとに用意される処理ルーチンを呼び出して実行する。一方、JITコンパイラ56は、バイトコードを事前にあるいは実行する直前にコンパイラを用いてマシン・コード58に変換してそれをCPUで実行する。

【0018】

JITコンパイラ56は、以下に詳細に述べる多次元配列オブジェクトの処理方法を実施する。ここでは、多次元配列の要素の処理が最適化できる条件を、多

次元配列の要素が格納されている最も下位の次元の配列（以下、基底配列）が連続したメモリ領域に割り当てられていることとする。その一例を図1に示す。多次元配列の要素の配置をこのような状態に保つことで、要素にアクセスする処理間の依存解析が可能となる。本実施例では、Javaバイトコードのmultianewarray命令が実行されるときに、基底配列が連続的に配置されるような多次元配列の生成を行い、その後、aastore 命令によってその多次元配列を構成する配列オブジェクト（図1参照）に書き込みが発生したときに処理最適化可能フラグを落とすような実装を行っている。以上に関する具体的な処理を1. 1～1. 2に示す。また、Javaはマルチスレッドでプロセスが実行されるため、これを考慮した特別の措置が必要である。これについては、1. 3で触れる。なお、1. 2では処理の手順が記されているが、実施にはその処理に対応した機械コードが生成されるものとする。最後に、1. 4において、aastore に処理を追加したときの実行時間の低下について議論する。

【0019】

1. 1：コンパイル時の処理（図5）

多次元配列の要素の処理が最適化できる条件を、多次元配列の基底配列が連続したメモリ領域に割り当てられていることとする。

1. 基底配列が連続したメモリ領域に割り当たられていることを前提とし、処理を最適化した機械コードを生成する。
2. 多次元配列の要素を処理する、最適化を行っていない機械コードを生成する。

【0020】

1. 2：実行時の処理（図6）

1. 2. 1：多次元配列生成時の処理

multianewarray命令が実行されたときに以下の処理を行う。

1. 多次元配列の全ての要素に必要なメモリサイズを計算する。
2. 計算したサイズのメモリを割り当てる。
3. 割り当てた領域に、基底配列を生成する。
4. 多次元配列を構成する配列オブジェクトの配列に必要なメモリサイズを計算

する。

5. 計算したサイズのメモリを割り当てる。
6. 割り当てた領域に、多次元配列を構成する配列オブジェクトの配列を作成する。
7. 6. で作成した配列の処理最適化可能フラグを立てる。
8. 6. で作成した配列に、上位の配列オブジェクトの配列へのポインタを加える。その一例を図2に示す。

【0021】

1. 2. 2 : 多次元配列を構成している配列オブジェクトへの書き込み時の処理
(図7)

aastore 命令が実行されたときに以下の処理を行う。

1. 書き込み対象となっている配列の処理最適化可能フラグを落とす。
2. 書き込み対象となっている配列の上位配列へのポインタを参照しながら、上位配列の処理最適化可能フラグを再帰的に落としていく。基底配列の連続性が保たれている下位配列に対しては、処理最適化可能フラグをそのままにしておく。その一例を図3に示す。
3. 以上のフラグの更新が終わった後、配列オブジェクトの内容を変更する。

【0022】

1. 2. 3 : 多次元配列アクセス時の処理 (図8)

1. 処理対象となっている多次元配列の処理最適化可能フラグを読み込む。
2. 処理対象に処理最適化フラグが立っている場合、最適化コードが実行されるようにする。
3. 処理対象に処理最適化フラグが立っておらず、対象配列に下位の次元が存在する場合、下位の次元に再帰的にこの処理を行う。
4. 処理対象に処理最適化フラグが立っておらず、対象配列に下位の次元が存在しない場合、非最適化コードが実行されるようにする。

【0023】

1. 2. 3. 1 : 具体例

以下に、多次元配列アクセスの具体例を示す。

.....

```
xA 100 100 =1.0;
```

.....

```
for i= 0 to 100
```

```
  for k= 0 to 100
```

```
    vB i =vB i +xA k i *vB i
```

この例では、ループに先立ってxA 100 100 をアクセスしているため、本来なら、ループ内でのxAのアクセスに例外が起こることはない。しかしながら、aastoreによる多次元配列の書き換えの可能性があるため、このような例では、xA 0, xA 1, ..., xA 100 を全てプライベートライズ（スレッドのローカルにコピーする）でもしない限り、最内ループにおけるxA k i のアクセスに対し毎回インデックスの範囲チェックを行わなければならない。一方、本発明を用いることで、このコードは以下のように改善される。

【0024】

.....

```
xA 100 100 =1.0;
```

.....

```
for i= 0 to 100
```

```
  if(IsOptimizable(xA)) {
```

```
    for k= 0 to 100
```

```
      vB i =vB i +xA k i *vB i ;// 範囲チェックなしのコードを生成
```

```
  } else {
```

```
    for k= 0 to 100
```

```
      vB i =vB i +xA k i *vB i ;// 範囲チェック付きのコードを生成
```

```
  }
```

上記のような最適化を行うことで、最内ループから条件分岐がなくなり、実行時間は大きく短縮される。

【0025】

1. 3 : マルチスレッド処理に対するサポート

マルチスレッド環境においては、同じ多次元配列の処理最適化可能フラグに対し、複数のプロセスが並列に読み込み／書き込みを行う可能性がある。本発明を利用したシステムが正しく動作するためには、このようなアクセスが発生した場合のサポートが必要になる。以下に、具体的な措置を示す。

1. 3. 1 : フラグが立っている配列と実際にアクセスする配列との同一性の保証

配列の要素が格納されているアドレスの取得とフラグの読み込みを個別のサイクルで行った場合、両サイクルの間で他のプロセスがこれらの値を書き換える可能性がある。この場合、処理最適化可能フラグの内容とそのフラグによって処理最適化可能性が指示される配列が一致しないので、本発明を利用したシステムがJavaの言語仕様に違反してしまう可能性がある。この問題は、一般的に、これらの走査をクリティカルセクションにすることで解決できるが、ここでは、本実施例に特有な性質を利用し、以下の方法で問題を解決する。

【0026】

1. 最適化ルーチンに入るときに、アクセスする配列の要素アドレス→フラグの順番で読み込みを行う。本実施例では、フラグは立っている状態から、立っていない状態へ一度しか変化しない。したがって、読み込みの時点でフラグが立っている場合、フラグ内容とフラグの指示対象は必ず一致していることが保証できる。また、フラグが立っていない場合は、配列にアクセスするコードは最適化されていないコードなので、問題が発生することはない。

(具体例)

....

for i= 0 to 100

if(IsOptimizable(xA)) {

/*最適化ルーチン*/

for k= 0 to 100

vB i =vB i +xA k i *vB i ;// 範囲チェックなし

} else

....

【0027】

例えば、先ほどのコードの最内ループでは、xAのアクセスは全て基底配列の先頭アドレスからの相対アドレスを用いて行われるような機械語が生成される。もし、フラグ→先頭アドレスの順番で読み込みを行ってしまうと、二つの読み込みの間にaastore による割り込み（例えば、xA 1 =null; のような）が起こった場合、Javaの言語仕様に対する違反がおきてしまう。一方、アドレス→フラグの順番で読み込みを行った場合は、そのような問題は起こらない。

【0028】

2. フラグ→アクセスする配列の要素アドレスの順番の書き込みを行う。この順番で走査を行うことで、フラグの書き込みとアドレスの変更の間に最適化コードが動作し始めても言語仕様に違反することがなくなる。

【0029】

3. 3次元以上の多次元配列では、フラグの書き込みが階層的に行われる。ここでは、Javaの言語仕様の保持のため、フラグの書き込みを上位から下位へ行う。

(具体例)

例えば、フラグを下位から上位に落としていくと、このプロセスの間に、上位次元の最適化コードが開始されてしまう可能性がある。このコードが下位次元の配列のフラグをチェックしないようなもの（例えば上記のようなコード）であると、Javaの言語仕様に違反してしまう危険がある。

【0030】

1. 3. 2 : ガーベッジコレクタの制御

あるプロセスAが、処理最適化可能性を持つ多次元配列に対し、要素のアドレスの計算をオブジェクト配列を使用せずに基底配列の連続性を利用して行っている場合、配列の処理の最中に、他のプロセスBがその多次元配列を構成する配列オブジェクトの配列を変更する可能性がある。Javaの仕様では、これらの操作が非同期メソッドで呼び出された場合、プロセスAは基底配列連続性を仮定した処理を行ってよい。しかし、プロセスBのオブジェクト配列の変更によって基底配列の一部がどこからも参照されなくなり、プロセスAの処理中に処理対象の配列

要素がガーベッジコレクタによって回収されてしまう可能性がある。この問題に対処するため、本実施例では、最適化コードによる処理が行われている間、スタック上に配列に対するダミーの参照を積む機械コードを生成することで、ガーベッジコレクタによる回収を防止する。

【0031】

1. 4 : astore に追加された処理による副作用に関する議論

以下の表1のようなプログラム例では、最も内側のループに対するバイトコードは表2のようになる。

【0032】

【表1】

プログラム例

```
public void sortObj(Obj[] obj){
    for(int i=0;i<obj.length;i++)
        for(int j=i;j<obj.length;j++)
            if(obj[i].data >= obj[j].data){
                Obj tmp = obj[i];
                obj[i] = obj[j];
                obj[j] = tmp;
            }
}
```

【0033】

【表2】

最内ループのバイトコード

Label1:	iload 2 <i>
aload 1 <obj[]>	aload 1 <obj[]>
iload 2 <i>	iload 3 <j>
aaload <obj[i]>	aaload <obj[j]>
getfield <data>	astore <obj[i] = obj[j]>
aload 1 <obj[]>	aload 1 <obj>
iload 3 <j>	iload 3 <j>
aaload <obj[j]>	aload 4 <tmp>
getfield <data>	astore <obj[j] = tmp>
if_icmplt Label2:	Label2:
aload 1 <obj[]>	iinc 3 1 <j++>
iload 2 <i>	aload 1 <obj>
aaload <obj[i]>	arraylength <obj.length>
astore 4 <tmp>	if_icmplt Label1:
aload 1 <obj[]>	

【0034】

この配列の生成時には、処理最適化可能フラグを立てられないので、ループ外でフラグを判定し、このフラグの内容でループをバージョニングすることも可能である。すなわち、以下の表3のようになる。なお、ここで副作用付きとは、aastore 命令実行時に本発明に関する処理を実施するものであり、副作用なしとは単純なaastore 命令の実行のみであることを示す。

【0035】

【表3】

```

if( IsNotOptimizable(obj) ){
    for(int i=0;i<obj.length;i++){
        for(int j=i;j<obj.length;j++){
            if(obj[i].data >= obj[j].data){
                Obj tmp = obj[i];
                obj[i] = obj[j]; // 副作用なしの aastore を生成
                obj[j] = tmp;    // 副作用なしの aastore を生成
            }
        }
    }
}else{
    for(int i=0;i<obj.length;i++){
        for(int j=i;j<obj.length;j++){
            if(obj[i].data >= obj[j].data){
                Obj tmp = obj[i];
                obj[i] = obj[j]; // 副作用付きの aastore を生成
                obj[j] = tmp;    // 副作用付きの aastore を生成
            }
        }
    }
}

```

【0036】

なお、最初のif文の条件がIsNotOptimizable(obj)である点に注意されたい。すなわち、最適化され得ないようなプログラムの場合には、aastore について本発明に関する処理を実施する必要はなく、最適化され得る状態の場合には、aastore について本発明の処理を実施する（else文以下）。

【0037】

【発明の効果】

以上の説明から明らかなように、本発明によれば、多次元配列の処理最適化可能フラグを導入し、多次元配列の要素構成の変更を監視しながら、同フラグが立っている配列に対して最適化されたコードを実行してよいことを保証しているため、仕様の変更を伴うことなく、多次元配列の処理最適化可能性を実行時に高速に判断し、処理最適化可能性がある判断された多次元配列を最適化されたコードで処理することにより、プログラムの実行速度を向上させることができる。その結果、従来は困難であったJavaなどにおける配列オブジェクトの配列で実現された多次元配列の処理の高速化が可能になる。

【図面の簡単な説明】

【図1】本発明における多次元配列オブジェクトの一例を説明するための図である。

【図2】本発明の多次元配列オブジェクトにおいてポインタの一例を説明するための図である。

【図3】本発明の多次元配列オブジェクトにおける書き込み時の処理最適化可能フラグの取り扱いの一例を説明するための図である。

【図4】本発明のシステム全体の概要図である。

【図5】本発明におけるコンパイル時の処理の一例を示すフローチャートである。

【図6】本発明における多次元配列生成時の処理の一例を示すフローチャートである。

【図7】本発明における多次元配列を構成している配列オブジェクトへの書き込み時の処理の一例を示すフローチャートである。

【図8】本発明における多次元配列アクセス時の処理の一例を示すフローチャートである。

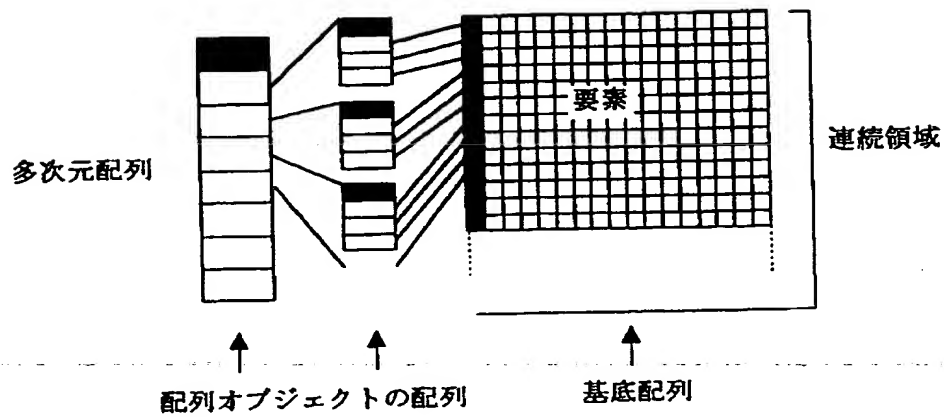
【符号の説明】

- 1 サーバ・コンピュータ
- 3 ネットワーク
- 5 クライアント・コンピュータ
- 52 Java VM

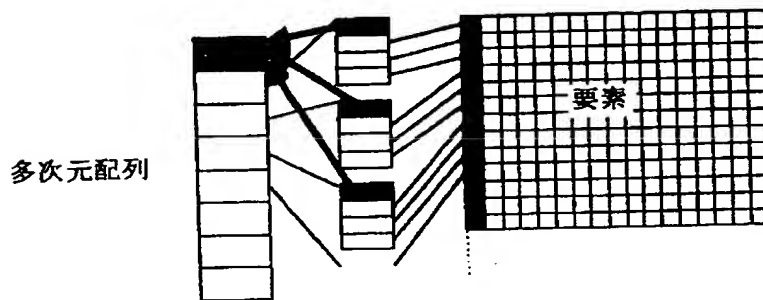
- 54 J a v a インタプリタ
- 56 J a v a J I T コンパイラ
- 58 マシンコード
- 60 ガベージ・コレクタ
- 53 O S
- 55 ハードウェア (CPU 及びメモリを含む)

【書類名】 図面

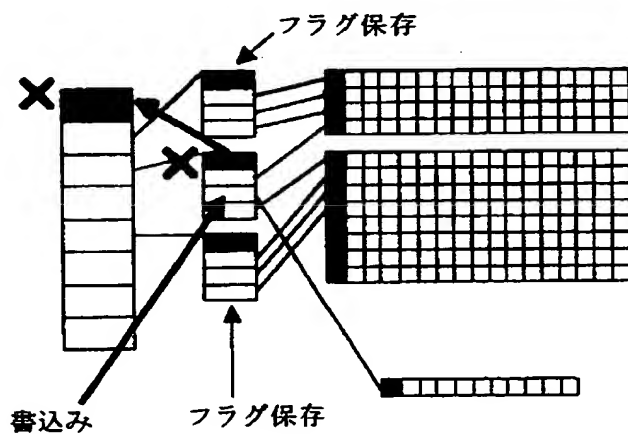
【図 1】



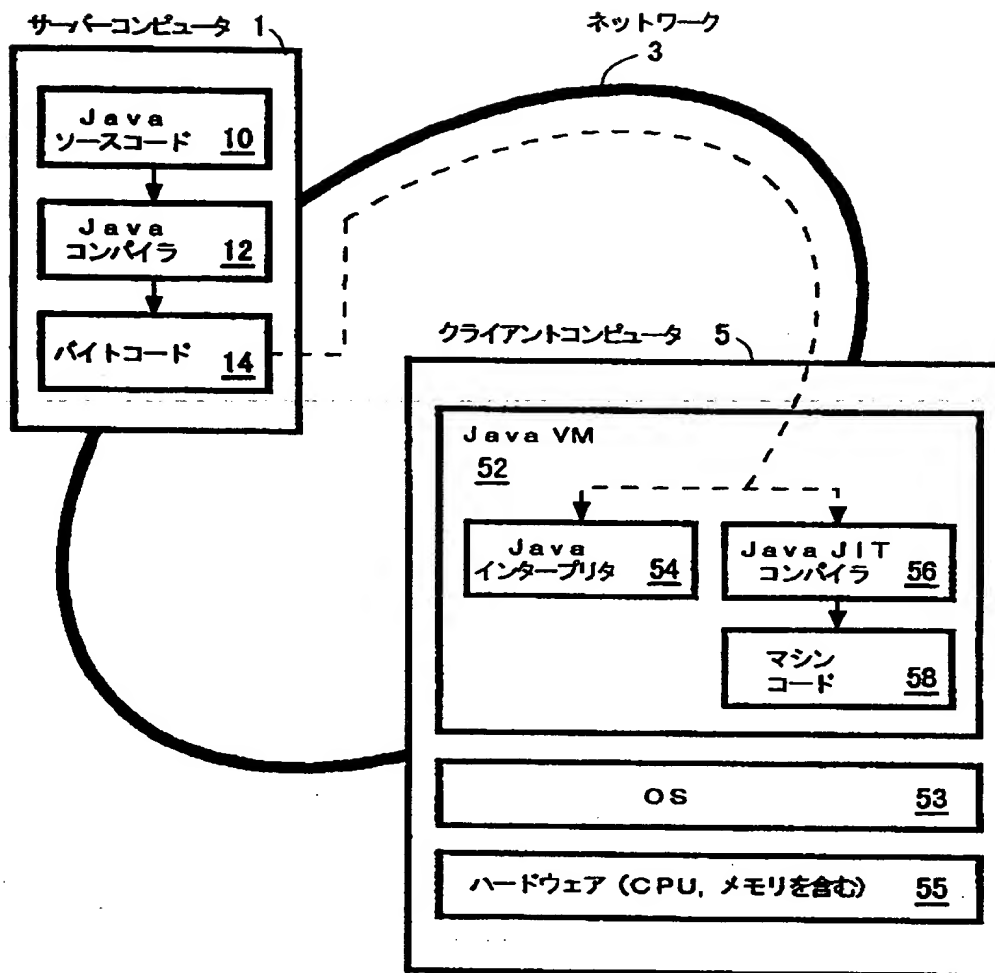
【図 2】



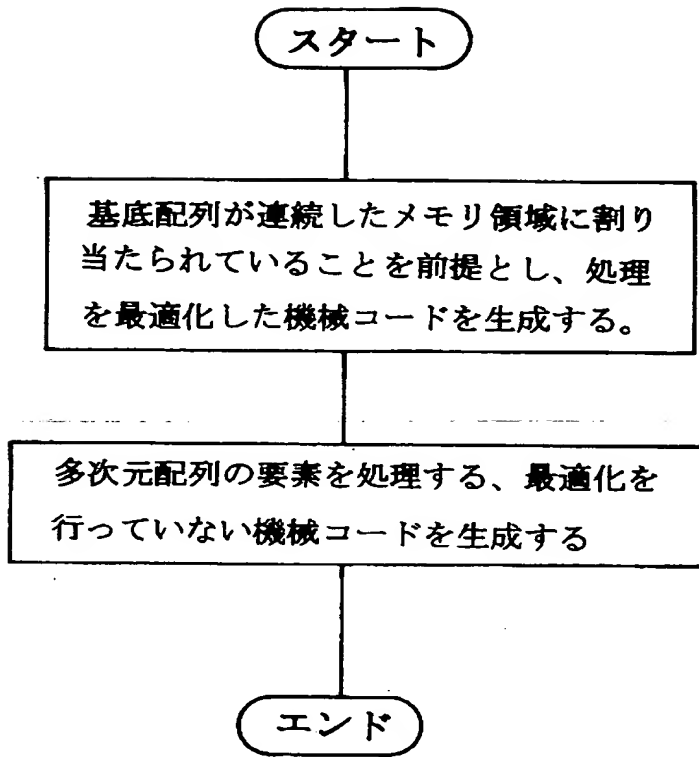
【図 3】



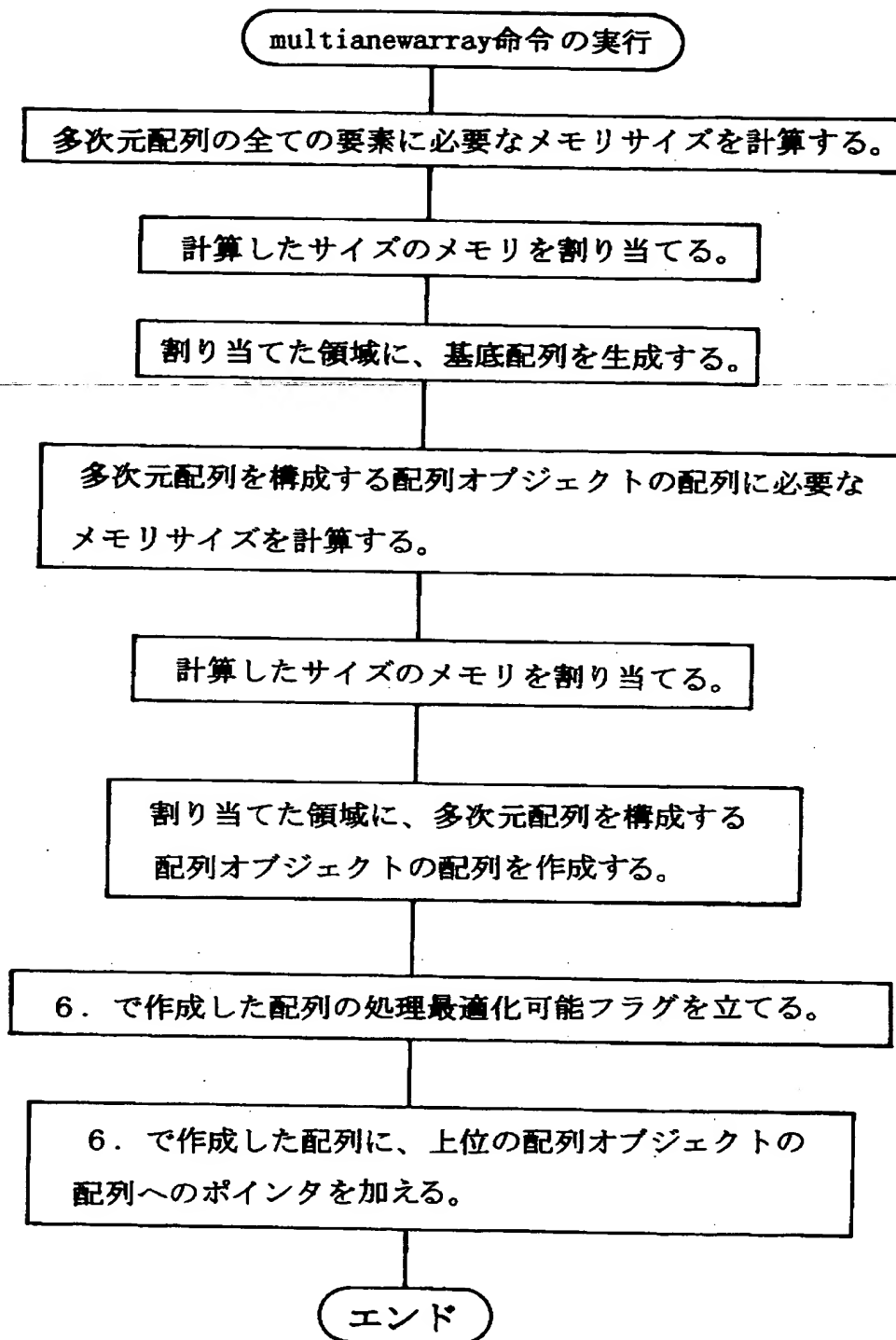
【図 4】



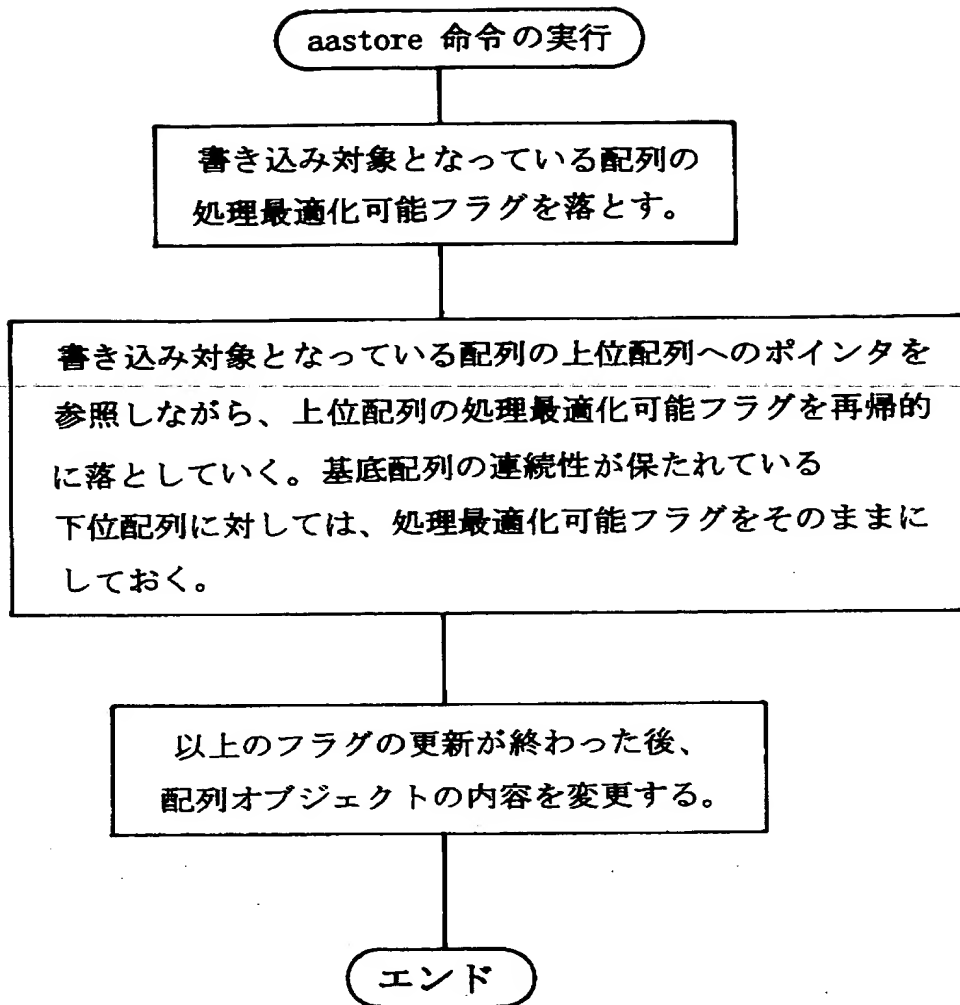
【図 5】



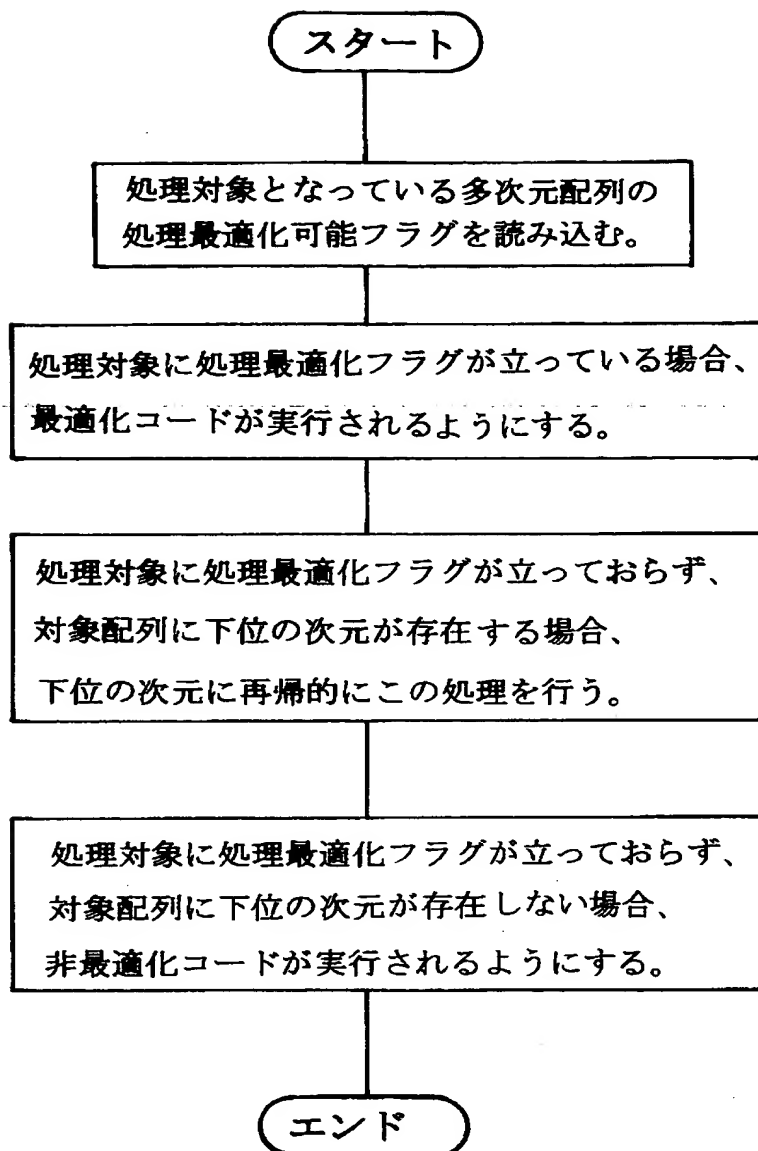
【図 6】



【図 7】



【図 8】



【書類名】 要約書

【要約】

【課題】 仕様の変更を伴うことなく、多次元配列の処理速度の向上を可能とする多次元配列オブジェクトの処理方法及び装置を提供する。

【解決手段】 本発明の多次元配列オブジェクトの処理方法は、多次元配列が配列オブジェクトの配列で実現された言語（例えばJava）における多次元配列オブジェクトの処理方法が対象となる。そして、多次元配列を構成している配列オブジェクトからなる多次元配列オブジェクトに、その多次元配列の要素に対する処理を最適化してもよいことを示す処理最適化可能フラグを付加情報として追加する。処理最適化可能フラグは、記憶装置（例えばメインメモリ）に格納される。その後、処理最適化可能フラグの状態に対応した機械コードの実行を行う。

【選択図】 図 1

出 願 人 履 歴 情 報

識別番号 [390009531]

1. 変更年月日 1990年10月24日

[変更理由] 新規登録

住 所 アメリカ合衆国10504、ニューヨーク州 アーモンク (番地なし)

氏 名 インターナショナル・ビジネス・マシーンズ・コーポレイション